下载APP

12 | 广播变量（一）：克制Shuffle，如何一招制胜！

2021-04-09 吴磊

Spark性能调优实战

进入课程 >

讲述：吴磊

时长 15:41　大小 14.38M

你好，我是吴磊。

在数据分析领域，数据关联（Joins）是 Shuffle 操作的高发区，二者如影随从。可以说，有 Joins 的地方，就有 Shuffle。

我们说过，面对 Shuffle，开发者应当“能省则省、能拖则拖”。我们已经讲过了怎么拖，拖指的就是，把应用中会引入 Shuffle 的操作尽可能地往后面的计算步骤去拖。那具体该怎么省呢？

在数据关联场景中，广播变量就可以轻而易举地省去 Shuffle。所以今天这一讲，我们就先说一说广播变量的含义和作用，再说一说它是如何帮助开发者省去 Shuffle 操作的。

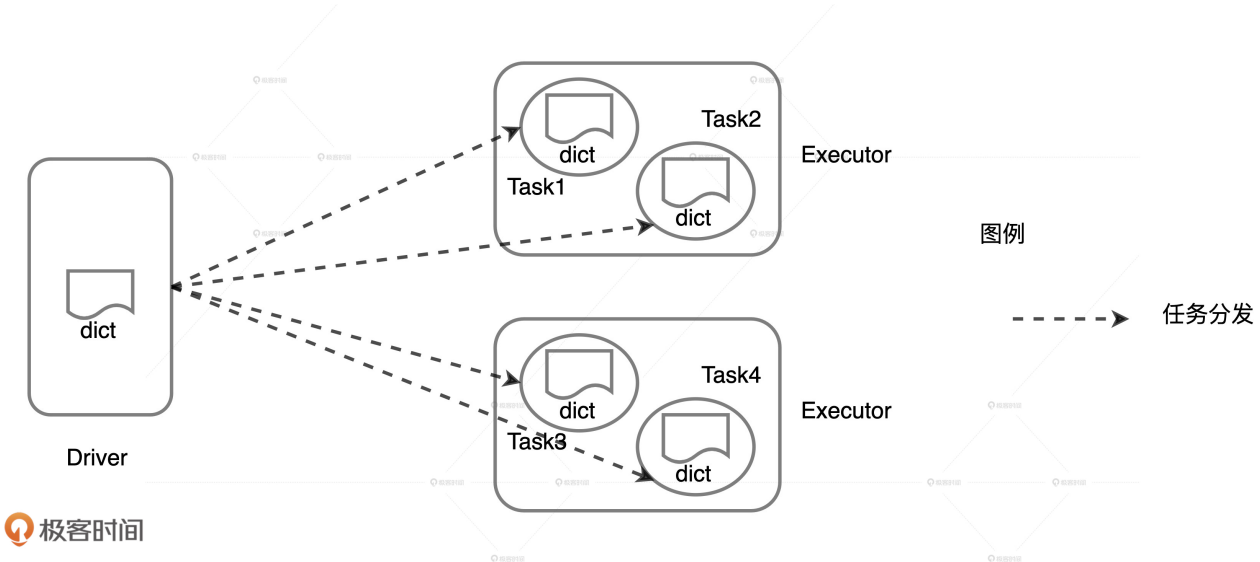
如何理解广播变量？

接下来，咱们借助一个小例子，来讲一讲广播变量的含义与作用。这个例子和 Word Count 有关，它可以说是分布式编程里的 Hello world 了，Word Count 就是用来统计文件中全部单词的，你肯定已经非常熟悉了，所以，我们例子中的需求增加了一点难度，我们要对指定列表中给定的单词计数。

复制代码

```
1 val dict = List(“spark”, “tune”)
2 val words = spark.sparkContext.textFile(“~/words.csv”)
3 val keywords = words.filter(word => dict.contains(word))
4 keywords.map((_, 1)).reduceByKey(_ + _).collect
```

按照这个需求，同学小 A 实现了如上的代码，一共有 4 行，我们逐一来看。第 1 行在 Driver 端给定待查单词列表 dict；第 2 行以 textFile API 读取分布式文件，内容包含一列，存储的是常见的单词；第 3 行用列表 dict 中的单词过滤分布式文件内容，只保留 dict 中给定的单词；第 4 行调用 reduceByKey 对单词进行累加计数。



数据结构dict随着Task一起分发到Executors

学习过调度系统之后，我们知道，第一行代码定义的 dict 列表连带后面的 3 行代码会一同打包到 Task 里面去。这个时候，Task 就像是一架架小飞机，携带着这些“行李”，飞往集群中不同的 Executors。对于这些“行李”来说，代码的“负重”较轻，可以忽略不计，而数据的负重占了大头，成了最主要的负担。

你可能会说：“也还好吧，dict 列表又不大，也没什么要紧的”。但是，如果我们假设这个例子中的并行度是 10000，那么，Driver 端需要通过网络分发总共 10000 份 dict 拷贝。这个时候，集群内所有的 Executors 需要消耗大量内存来存储这 10000 份的拷贝，对宝贵的网络和内存资源来说，这已经是一笔不小的浪费了。更何况，如果换做一个更大的数据结构，Task 分发所引入的网络与内存开销会更可怕。

换句话说，统计计数的业务逻辑还没有开始执行，Spark 就已经消耗了大量的网络和存储资源，这简直不可理喻。因此，我们需要对示例中的代码进行优化，从而跳出这样的窘境。

但是，在着手优化之前，我们不妨先来想一想，现有的问题是什么，我们要达到的目的是什么。结合刚刚的分析，我们不难发现，**Word Count 的核心痛点在于，数据结构的分发和存储受制于并行，并且是以 Task 为粒度的，因此往往频次过高。痛点明确了，调优的目的也就清晰了，我们需要降低数据结构分发的频次。**

要达到这个目的，我们首先想到的就是降低并行度。不过，牵一发而动全身，并行度一旦调整，其他与 CPU、内存有关的配置项都要跟着适配，这难免把调优变复杂了。实际上，要降低数据结构的分发频次，我们还可以考虑广播变量。

广播变量是一种分发机制，它一次性封装目标数据结构，以 Executors 为粒度去做数据分发。换句话说，在广播变量的工作机制下，数据分发的频次等同于集群中的 Executors 个数。通常来说，集群中的 Executors 数量都远远小于 Task 数量，相差两到三个数量级是常有的事。那么，对于第一版的 Word Count 实现，如果我们使用广播变量的话，会有哪些变化呢？

代码的改动很简单，主要有两个改动：第一个改动是用 broadcast 封装 dict 列表，第二个改动是在访问 dict 列表的地方改用 broadcast.value 替代。

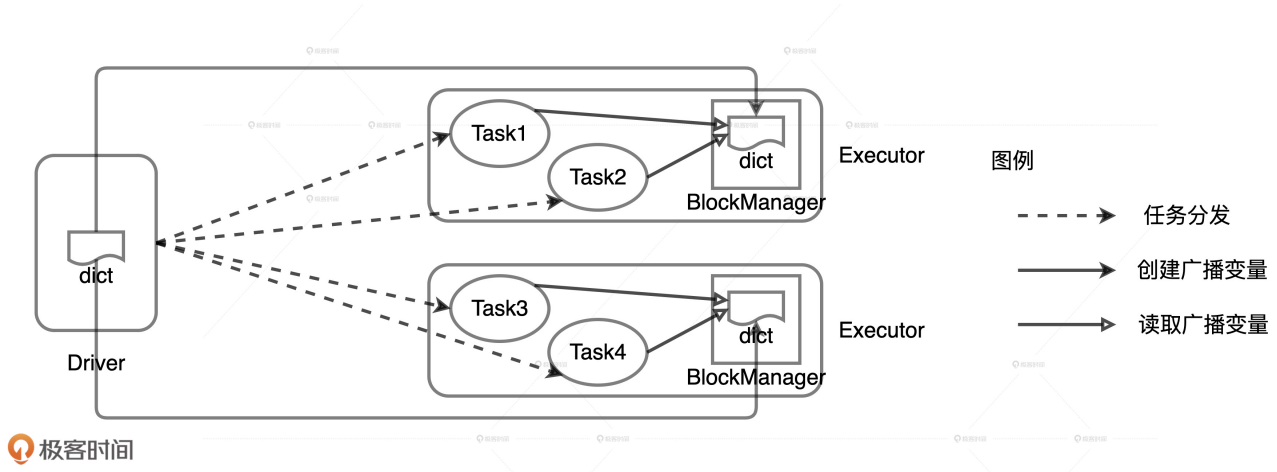
复制代码

```
1 val dict = List(“spark”, “tune”)
2 val bc = spark.sparkContext.broadcast(dict)
3 val words = spark.sparkContext.textFile(“~/words.csv”)
4 val keywords = words.filter(word => bc.value.contains(word))
5 keywords.map((_, 1)).reduceByKey(_ + _).collect
```

你可能会说：“这个改动看上去也没什么呀！”别着急，我们先来分析一下，改动之后的代码在运行时都有哪些变化。

在广播变量的运行机制下，封装成广播变量的数据，由 Driver 端以 Executors 为粒度分发，每一个 Executors 接收到广播变量之后，将其交给 BlockManager 管理。由于广播变量携带的数据已经通过专门的途径存储到 BlockManager 中，因此分发到 Executors 的 Task 不需要再携带同样的数据。

这个时候，你可以把广播变量想象成一架架专用货机，专门为 Task 这些小飞机运送“大件行李”。Driver 与每一个 Executors 之间都开通一条这样的专用货机航线，统一运载负重较大的“数据行李”。有了专用货机来帮忙，Task 小飞机只需要携带那些负重较轻的代码就好了。等这些 Task 小飞机在 Executors 着陆，它们就可以到 Executors 的公用仓库 BlockManager 里去提取它们的“大件行李”。



用广播变量封装dict列表

总之，在广播变量的机制下，dict 列表数据需要分发和存储的次数锐减。我们假设集群中有 20 个 Executors，不过任务并行度还是 10000，那么，Driver 需要通过网络分发的 dict 列表拷贝就会由原来的 10000 份减少到 20 份。同理，集群范围内所有 Executors 需要存储的 dict 拷贝，也由原来的 10000 份，减少至 20 份。这个时候，引入广播变量后的开销只是原来 Task 分发的 1/500！

广播分布式数据集

那在刚刚的示例代码中，广播变量封装的是 Driver 端创建的普通变量：字符串列表。除此之外，**广播变量也可以封装分布式数据集。**

我们来看这样一个例子。在电子商务领域中，开发者往往用事实表来存储交易类数据，用维度表来存储像物品、用户这样的描述性数据。事实表的特点是规模庞大，数据体量随着业务的发展不断地快速增长。维度表的规模要比事实表小很多，数据体量的变化也相对稳定。

假设用户维度数据以 Parquet 文件格式存储在 HDFS 文件系统中，业务部门需要我们读取用户数据并创建广播变量以备后用，我们该怎么做呢？很简单，几行代码就可以搞定！

复制代码

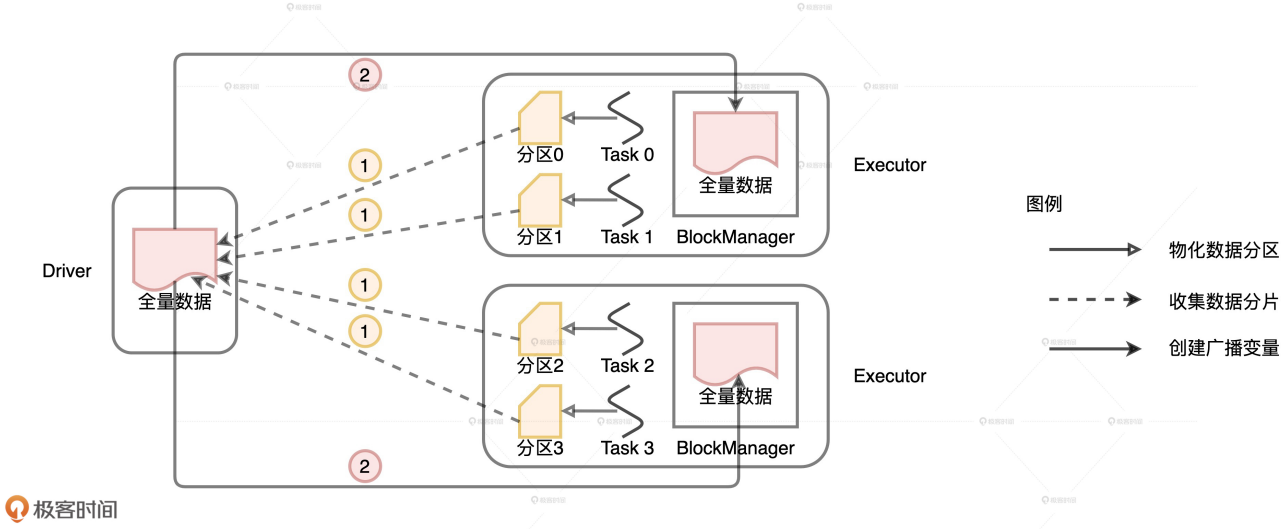
```
1 val userFile: String = “hdfs://ip:port/rootDir/userData”
2 val df: DataFrame = spark.read.parquet(userFile)
3 val bc_df: Broadcast[DataFrame] = spark.sparkContext.broadcast(df)
```

首先，我们用 Parquet API 读取 HDFS 分布式数据文件生成 DataFrame，然后用 broadcast 封装 DataFrame。从代码上来看，这种实现方式和封装普通变量没有太大差别，它们都调用了 broadcast API，只是传入的参数不同。

但如果不从开发的视角来看，转而去观察运行时广播变量的创建过程的话，我们就会发现，分布式数据集与普通变量之间的差异非常显著。

从普通变量创建广播变量，由于数据源就在 Driver 端，因此，只需要 Driver 把数据分发到各个 Executors，再让 Executors 把数据缓存到 BlockManager 就好了。

但是，从分布式数据集创建广播变量就要复杂多了，具体的过程如下图所示。



从分布式数据集创建广播变量的过程

与普通变量相比，分布式数据集的数据源不在 Driver 端，而是来自所有的 Executors。Executors 中的每个分布式任务负责生产全量数据集的一部分，也就是图中不同的数据分区。因此，步骤 1 就是 **Driver 从所有的 Executors 拉取这些数据分区，然后在本地构建全量数据。**步骤 2 与从普通变量创建广播变量的过程类似。 **Driver 把汇总好的全量数据分发给各个 Executors，Executors 将接收到的全量数据缓存到存储系统的 BlockManager 中。**

不难发现，相比从普通变量创建广播变量，从分布式数据集创建广播变量的网络开销更大。原因主要有二：一是，前者比后者多了一步网络通信；二是，前者的数据体量通常比后者大很多。

如何用广播变量克制 Shuffle?

你可能会问：“Driver 从 Executors 拉取 DataFrame 的数据分片，揉成一份全量数据，然后再广播出去，抛开网络开销不说，来来回回得费这么大劲，图啥呢？”这是一个好问题，因为以广播变量的形式缓存分布式数据集，正是克制 Shuffle 杀手锏。

Shuffle Joins

为什么这么说呢？我还是拿电子商务场景举例。有了用户的数据之后，为了分析不同用户的购物习惯，业务部门要求我们对交易表和用户表进行数据关联。这样的数据关联需求在数据分析领域还是相当普遍的。

复制代码

```
1 val transactionsDF: DataFrame = _
2 val userDF: DataFrame = _
3 transactionsDF.join(userDF, Seq(“userID”), “inner”)
```

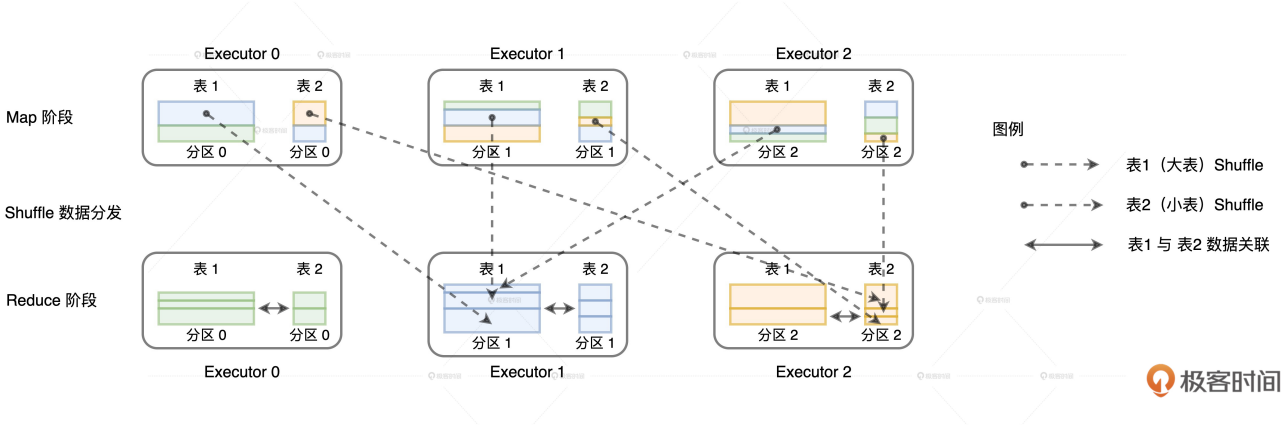
因为需求非常明确，同学小 A 立即调用 Parquet 数据源 API，读取分布式文件，创建交易表和用户表的 DataFrame，然后调用 DataFrame 的 Join 方法，以 userID 作为 Join keys，用内关联（Inner Join）的方式完成了两表的数据关联。

在分布式环境中，交易表和用户表想要以 userID 为 Join keys 进行关联，就必须要确保一个前提：交易记录和与之对应的用户信息在同一个 Executors 内。也就是说，如果用户黄

小乙的购物信息都存储在 Executor 0，而个人属性信息缓存在 Executor 2，那么，在分布式环境中，这两种信息必须要凑到同一个进程里才能实现关联计算。

在不进行任何调优的情况下，Spark 默认采用 Shuffle Join 的方式来做到这一点。Shuffle Join 的过程主要有两步。

第一步就是对参与关联的左右表分别进行 Shuffle，Shuffle 的分区规则是先对 Join keys 计算哈希值，再把哈希值对分区数取模。由于左右表的分区数是一致的，因此 Shuffle 过后，一定能够保证 userID 相同的交易记录和用户数据坐落在同一个 Executors 内。



Shuffle Join中左右表的数据分发

Shuffle 完成之后，**第二步就是在同一个 Executors 内，Reduce task 就可以对 userID 一致的记录进行关联操作**。但是，由于交易表是事实表，数据体量异常庞大，对 TB 级别的数据进行 Shuffle，想想都觉得可怕！因此，上面对两个 DataFrame 直接关联的代码，还有很大的调优空间。我们该怎么做呢？话句话说，对于分布式环境中的数据关联来说，要想确保交易记录和与之对应的用户信息在同一个 Executors 中，我们有没有其他办法呢？

克制 Shuffle 的方式

还记得之前业务部门要求我们把用户表封装为广播变量，以备后用吗？现在它终于派上用场了！

复制代码

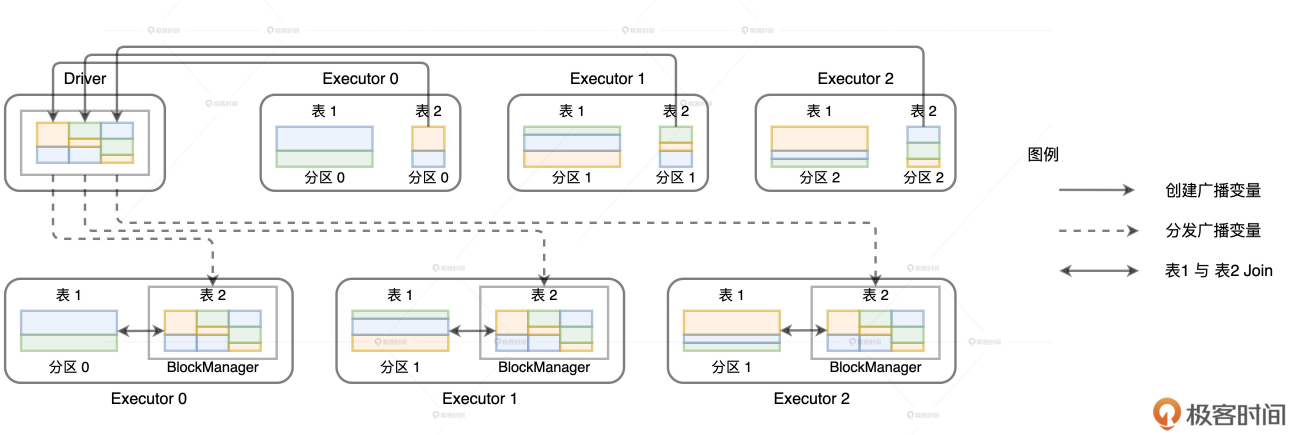
```
import org.apache.spark.sql.functions.broadcast

val transactionsDF: DataFrame = _
val userDF: DataFrame = _

val bcUserDF = broadcast(userDF)
```

```
7 transactionsDF.join(bcUserDF, Seq("userID"), "inner")
8
```

Driver 从所有 Executors 收集 userDF 所属的所有数据分片，在本地汇总用户数据，然后给每一个 Executors 都发送一份全量数据的拷贝。既然每个 Executors 都有 userDF 的**全量数据**，这个时候，交易表的数据分区待在原地、保持不动，就可以轻松地关联到一致的用户数据。如此一来，我们不需要对数据体量巨大的交易表进行 Shuffle，同样可以在分布式环境中，完成两张表的数据关联。



Broadcast Join将小表广播，避免大表Shuffle

利用广播变量，我们成功地避免了海量数据在集群内的存储、分发，节省了原本由 Shuffle 引入的磁盘和网络开销，大幅提升运行时执行性能。当然，采用广播变量优化也是有成本的，毕竟广播变量的创建和分发，也是会带来网络开销的。但是，相比大表的全网分发，小表的网络开销几乎可以忽略不计。这种小投入、大产出，用极小的成本去博取高额的性能收益，真可以说是“四两拨千斤”！

小结

在数据关联场景中，广播变量是克制 Shuffle 的杀手锏。掌握了它，我们就能以极小的成本，获得高额的性能收益。关键是我们要掌握两种创建广播变量的方式。

第一种，从普通变量创建广播变量。在广播变量的运行机制下，普通变量存储的数据封装成广播变量，由 Driver 端以 Executors 为粒度进行分发，每一个 Executors 接收到广播变量之后，将其交由 BlockManager 管理。

第二种，从分布式数据集创建广播变量，这就要比第一种方式复杂一些了。第一步，Driver 需要从所有的 Executors 拉取数据分片，然后在本地构建全量数据；第二步，

Driver 把汇总好的全量数据分发给各个 Executors，Executors 再将接收到的全量数据缓存到存储系统的 BlockManager 中。

结合这两种方式，我们在做数据关联的时候，把 Shuffle Joins 转换为 Broadcast Joins，就可以用小表广播来代替大表的全网分发，真正做到克制 Shuffle。

每日一练

1. Spark 广播机制现有的实现方式是存在隐患的，在数据量较大的情况下，Driver 可能会成为瓶颈，你能想到更好的方式来重新实现 Spark 的广播机制吗？（提示：SPARK-17556）
2. 在什么情况下，不适合把 Shuffle Joins 转换为 Broadcast Joins?

期待在留言区看到你的思考和答案，我们下一讲见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | Shuffle的工作原理：为什么说Shuffle是一时无两的性能杀手?

下一篇 13 | 广播变量（二）：有哪些途径让Spark SQL选择Broadcast Joins?

精选留言 (10)

写留言

Bennan
2021-04-09

1. 改成由driver获取到数据分布，然后通知各个executor之间进行拉取，这样可以利用多个executor网络，避免只有driver组装以后再一个一个发送效率过低

2.当两个需要join的数据集都很大时，使用broadcast join需要将一个很大的数据集进行网络分发多次，已经远超出了shuffle join需要传输的数据

展开

作者回复: Perfect！满分💯，两道题答的都很好～

9

Jack
2021-04-09

老师，对于第1题，看了下spark的源码，目前Broadcast只有一个实现类TorrentBroadcast，看代码的注释，这个类通过使用类似Bit-torrent协议的方法解决了Driver成为瓶颈的问题。目前Spark还会存在广播变量的数据太大造成Driver成为瓶颈的问题吗？

/**...
展开

作者回复: 非常赞哈👍哈～ 凡是看源码的同学，都先给个赞～

这块非常值得探讨。我的理解是这样的，代码层面，spark确实已经有code在尝试用p2p的方式来分发广播变量，从而减轻driver负担。

但是，据我观察，这部分代码尚未生效。细节可以参考这个ticket：【Executor side broadcast for broadcast joins】https://issues.apache.org/jira/browse/SPARK-17556，看上去还是进行中的状态。

另外，从代码看，目前还是先用collect拉到driver，然后再分发出去：

BroadcastExchangeExec中的relationFuture用于获取广播变量内容
在relationFuture内部：
1. 先是调用executeCollectIterator生成内容relation；
   其中，executeCollectIterator调用collect把结果集收集到driver端
2. 然后用sparkContext.broadcast(relation)，把生成好的内容广播到各个Executors
并没有看到哪里从Executors拉取数据分片、来减轻driver负载。

并且，这里还有提示driver内存不够的exception：
new OutOfMemoryError("Not enough memory to build and broadcast the table to all " + "worker nodes. As a workaround, you can either disable broadcast by setting " + s"${SQLConf.AUTO_BROADCASTJOIN_THRESHOLD.key} to -1 or increase the spark " + s"driver memory by setting ${SparkLauncher.DRIVER_MEMORY} to a higher value.").initCause(oe.getCause))

你贴的这段代码确实在尝试用片2片，不过，需要仔细看看，它在哪里调用，被谁调用～

2 3

Geek_d794f8
2021-04-10

老师有两个问题请教一下：

1.文中提到两个表join，两个表数据量相差很大呀，为什么他们的的分区数是一致的，而且分区数不是根据hadoop的切片规则去划分的吗？
2.广播join不是默认开启的吗，好像小表默认10M；还需像文中代码val bcUserDF = broadcast(userDF)这样声明吗？...

展开

作者回复: 思考的很深入，赞一个~

1. 我理解，你问的是这句吧？“由于左右表的分区数是一致的，因此 Shuffle 过后，一定能够保证 userID 相同的交易记录和用户数据坐落在同一个 Executors 内。” HadoopRDD的分区数、或者说并行度，确实是由HDFS文件系统决定的；但是，Shuffle过后，每个分布式数据集的并行度，就由参数spark.sql.shuffle.partitions来决定了，这个咱们在配置项哪一讲说过哟~ 因此，如果你没有手工用repartition或是Coalesce去调整并行度，默认情况下，大家Shuffle过后（在Reduce阶段）都是这个并行度。

2. 默认确实是开启的，默认值确实也是10MB，但是，这个10MB太太太太太太（太 x N）小了！很多小表其实都超过了这个阈值，因此，如果你懒得去调整这个参数，可以直接用broadcast(userDF)这种强制广播的方式，省时省力，比较方便~

1

斯盖丸
2021-05-05

老师我生产中为啥从没有遇到过10000并行度那么大的stage，可能我公司比较小吧，集群最多才100多个核，多数时才几百个任务，最多时也才2000多个任务。这健康吗？

展开

作者回复: 这里其实有两个容易混淆的概念哈~

一个是并行度，并行度其实是从数据角度出发，表示的是你的分布式数据集划分的粒度，再直白点说，它和分区数是等效的。因此，它其实跟你集群有多少Executors，每个Executors配置了多少cores，没有关系~

第二个是并发度，或者叫Executors线程池大小，也就是你用spark.executor.cores类似的参数，给Executors指定的cores资源。它限制了在同一时间，你的Executors中最多同时能跑多少个任务。Executors并发度乘以集群中的Executors数量，其实就是你集群的并发处理能力，很多地方也叫并行处理能力。其实蛋疼的地方在于，不同的作者、不同的上下文，并发和并行这两个词，总是混用。所以也就造成大家都比较困惑。

咱们在配置项第一讲，其实就在尝试厘清、约定这两个词的定义，一来方便大家理解，二来方便后续讨论。

所以，回答你的问题，其实没什么不健康的哈~ 10000并行度，意味着10000个分区的分布式数据集，这个应该不难见到。另外100个cores的集群，其实也不算小了~ 不过你说的2000任务我没有get到，不知道是2000并行度，还是2000的集群并发。如果是2000集群并发的话，这个数和100cores对不上。这意味着你的每个core需要20个超线程，哈哈，目前还没有这么给力的CPU。一般CPU也就2个超线程。

小灵芝
2021-04-30

“第一步就是对参与关联的左右表分别进行 Shuffle，Shuffle 的分区规则是先对 Join keys 计算哈希值，再把哈希值对分区数取模。由于左右表的分区数是一致的，因此 Shuffle 过后，一定能够保证 userID 相同的交易记录和用户数据坐落在同一个 Executors 内。”

老师请问什么叫吧哈希值对分区取模？取模是什么意思？...

展开

作者回复: 举个例子哈，比如你的分区数，是默认的200个。那么拿userId = '黄小乙'来说，它的哈希值假设是201，那么对200取模（数学运算：mod、modulo）就是1，那么凡是userId = '黄小乙'的数据记录，就都被存储到#1号数据分区，其他userId以此类推。

七
2021-04-18

老师，问一下 从hdfs加载parquet文件时，例如spark.read.parquet("xx")，是怎么确定分区数的呢？阅读源码DataFrameReader没有找到相关配置，希望老师给指条路，解答一下。

展开

作者回复: Data Source的并行度，取决于分布式文件本身，比如HDFS文件系统中的源数据，它本身的并行度，就是Spark构建的HadoopRDD的初始并行度。比如一个文件，在HDFS有1204个分片，那么有它构建的HadoopRDD的初始并行度，就是1024。

这一点，通过Spark UI也可以看出来哈~ 比如你在执行数据源读取的时候，发现Spark UI上显示：“238/1024”，这意味着你的源数据有1024个分片，Spark总共需要1024个task来完成数据读取，其中已经完成了238个任务，也就是成功读取了238个分区数据。

辰
2021-04-15

老师，您好，问题一和问题二有没有一种矛盾在里面，问题一是数据量大，如何使用广播，问题二我知道的是，当join的表数据量都很大时就不适合广播了，一是因为网络开销，二是因为executor内存不够大不足以存储大数据量，还有老师，能不能说一下广播的阈值多少合适，这个和其他的参数也有关系，比如整体的参数如何设置，能不能讲一下，executor数量，core的数量，分区大小以及内存大小设置

展开

作者回复: 这些参数之间的关系和设置，其实我们在配置项那两讲有过详细的展开哈~ 关于广播阈值的问题，就我的经验，平时工作中，2GB左右，是个不错的选择。当然，所有这些参数的设置，都没有绝对值，否则咱们就不需要调优了，大家都用一套绝对值就好了。就是因为大家的场景、部署、情况各自不同，所以才需要调优，而调优的原则，就可以follow配置项那两讲去操作。

比如你现在让我给出一个最佳组合，其实这个命题是不成立的，因为你需要给定一个场景，我才能具体说一个什么样的作业，在你的部署下，最佳的配置组合是什么。所以，调优，是以场景为基础的哈，也就是咱们只能就事论事，但这不代表我们就是瞎蒙，相反，我们设置不同的参数组合是基于一些基本原则的，而这些原则，就是配置项那两讲的内容所在~

1

Jack
2021-04-09

老师，第2题@Bennan的回答有疑惑。为什么当两个需要join的数据集都很大时，broadcast join会超出shuffle join需要传输的数据。
假设有A，B两表，各自大小都是100G，有4个executor，每个executor有4个task对应4个分区，总共16个task。，把shuffle的过程用二分图刻画，两边都各自有16个顶点，边有16*16条。...

展开

作者回复: 思考的很深入，先赞一个~ 我的理解是这样的，shuffle分发的，是部分数据。广播分发的，是全量数据。当然，像你说的，广播的粒度大，是executor level。shuffle粒度小，是task level。如果全量数据很大，即使能塞进广播变量，即便是executor粒度，我觉得它的开销，也远大于它的收益，甚至还有driver端oom的风险，所以这种情况，还不如采用shuffle join。 当然，这么说比较笼统，并没有精确的开销和收益的计算。这里其实更多的，是想强调广播不是银弹，不要滥用广播。

3

斯盖丸
2021-04-09

原来小表和大表join是节省了大表的shuffle，不然大表只能根据join的列在所有机器上重新分布一遍，现在懂了

展开

作者回复: 是的，以小博大

2021-04-09

Executor端的Broadcast这个功能，目前的spark版本是不是没有这个功能呀?

展开

作者回复: code其实有，但是就我理解，出于各种原因，并没有生效，具体细节可以参考给Jack的回复哈～